

Incremental 2-Edge-Connectivity in Directed Graphs

Loukas Georgiadis¹, Giuseppe F. Italiano², and Nikos Parotsidis²

¹University of Ioannina, Greece. loukas@cs.uoi.gr

²Università di Roma “Tor Vergata”, Italy. giuseppe.italiano@uniroma2.it,
nikos.parotsidis@uniroma2.it

July 26, 2016

Abstract

In this paper, we initiate the study of the dynamic maintenance of 2-edge-connectivity relationships in directed graphs. We present an algorithm that can update the 2-edge-connected blocks of a directed graph with n vertices through a sequence of m edge insertions in a total of $O(mn)$ time. After each insertion, we can answer the following queries in asymptotically optimal time:

- Test in constant time if two query vertices v and w are 2-edge-connected. Moreover, if v and w are not 2-edge-connected, we can produce in constant time a “witness” of this property, by exhibiting an edge that is contained in all paths from v to w or in all paths from w to v .
- Report in $O(n)$ time all the 2-edge-connected blocks of G .

To the best of our knowledge, this is the first dynamic algorithm for 2-connectivity problems on directed graphs, and it matches the best known bounds for simpler problems, such as incremental transitive closure.

1 Introduction

The design of dynamic graph algorithms is one of the classic areas in theoretical computer science. In this setting, the input of a graph problem is being changed via a sequence of updates, such as edge insertions and deletions. A dynamic graph algorithm aims at updating efficiently the solution of a problem after each update, faster than recomputing it from scratch. A dynamic graph problem is said to be *fully dynamic* if the update operations include both insertions and deletions of edges, and it is said to be *partially dynamic* if only one type of update, either insertions or deletions, is allowed. More specifically, a dynamic graph problem is said to be *incremental* (resp., *decremental*) if only insertions (resp., deletions) are allowed.

In this paper, we present new incremental algorithms for 2-edge connectivity problems on directed graphs (digraphs). Before defining the problem, we first review some definitions. Let $G = (V, E)$ be a digraph. G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* (in short *SCCs*) of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are *strongly connected* if they belong to the same SCC of G . An edge of G is a *strong bridge* if its removal increases the number of SCCs. Let G be strongly connected: G is *2-edge-connected* if it has no strong bridges. The *2-edge-connected components* of G are its maximal 2-edge-connected subgraphs. Two vertices $u, v \in V$ are said to

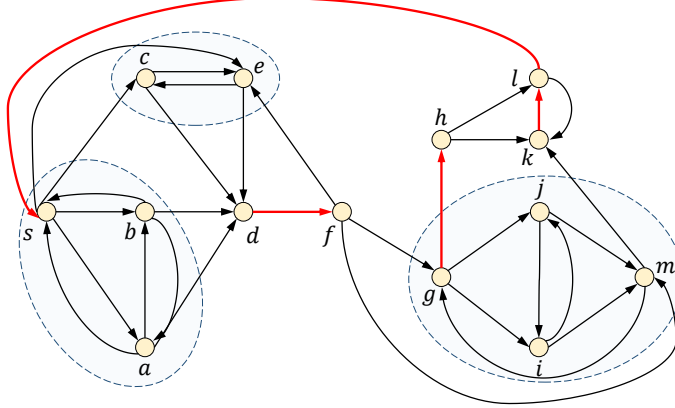


Figure 1: The 2-edge-connected blocks of a digraph G . Strong bridges of G are shown red. (Better viewed in color.)

be 2-edge-connected, denoted by $u \leftrightarrow_{2e} v$, if there are two edge-disjoint directed paths from u to v and two edge-disjoint directed paths from v to u . (Note that a path from u to v and a path from v to u need not be edge-disjoint). A 2-edge-connected block of a digraph $G = (V, E)$ is defined as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ for all $u, v \in B$. Figure 1 illustrates the 2-edge-connected blocks of a digraph.

We remark that in digraphs 2-vertex and 2-edge connectivity have a much richer and more complicated structure than in undirected graphs. To see this, observe that, while in undirected graphs blocks are exactly the same as components, in digraphs there is a substantial difference between those two notions. In particular, the edge-disjoint paths that make two vertices 2-edge-connected in a block might use vertices that are outside of that block, while in a component those paths must lie completely inside that component. In other words, two vertices that are 2-edge-connected (and thus in the same 2-edge-connected block) may lie in different 2-edge-connected components (e.g., vertices i and j in Figure 1, each of them being in a 2-edge-connected component by itself). As a result, 2-connectivity problems on digraphs appear to be much harder than on undirected graphs. For undirected graphs it has been known for over 40 years how to compute the 2-edge- and 2-vertex-connected components in linear time [36]. In the case of digraphs, however, only $O(mn)$ algorithms were known (see e.g., [27, 28, 31, 33]). It was shown only recently how to compute the 2-edge- and 2-vertex-connected blocks in linear time [14, 15], and the best current bound for computing the 2-edge- and the 2-vertex-connected components is $O(n^2)$ [20].

Our Results. In this paper, we initiate the study of the dynamic maintenance of 2-edge-connectivity relationships in directed graphs. We present an algorithm that can update the 2-edge-connected blocks of a digraph G with n vertices through a sequence of m edge insertions in a total of $O(mn)$ time. After each insertion, we can answer the following queries in asymptotically optimal time:

- Test in constant time if two query vertices v and w are 2-edge-connected. Moreover, if v and w are not 2-edge-connected, we can produce in constant time a “witness” of this property, by exhibiting an edge that is contained in all paths from v to w or in all paths from w to v .

- Report in $O(n)$ time all the 2-edge-connected blocks of G .

Ours is the first dynamic algorithm for 2-connectivity problems on digraphs, and it matches the best known bounds for simpler problems, such as incremental transitive closure [25]. This is a substantial improvement over the $O(m^2)$ simple-minded algorithm, which recomputes the 2-edge-connected blocks from scratch after each edge insertion.

Related Work. Many efficient algorithms for several dynamic graph problems have been proposed in the literature, including dynamic connectivity [22, 24, 34, 41], minimum spanning trees [9, 12, 23, 24], edge/vertex connectivity [9, 24] on undirected graphs, and transitive closure [8, 21, 29] and shortest paths [7, 29, 42] on digraphs. Once again, dynamic problems on digraphs appear to be harder than on undirected graphs. Indeed, most of the dynamic algorithms on undirected graphs have polylog update bounds, while dynamic algorithms on digraphs have higher polynomial update bounds. The hardness of dynamic algorithms on digraphs has been recently supported also by conditional lower bounds [1].

Our Techniques. Known algorithms for computing the 2-edge-connected blocks of a digraph G [14, 17] hinge on properties that seem very difficult to dynamize. The algorithm in [14] uses very complicated data structures based on 2-level auxiliary graphs. The loop nesting forests used in [17] depends heavily on an underlying dfs tree of the digraph, and the incremental maintenance of dfs trees on general digraphs is still an open problem (incremental algorithms are known only for the special case of DAGs [11]). Despite those inherent difficulties, we find a way to bypass loop nesting forests by suitably combining the approaches in [14, 17] in a novel framework, which is amenable to dynamic implementations. Another complication is that, although our problem is incremental, strong bridges may not only be deleted but also added (when a new SCC is formed). As a result, our data structures undergo a fully dynamic repertoire of updates, which is known to be harder. By organizing carefully those updates, we are still able to obtain the desired bounds.

2 Dominator trees and 2-edge-connected blocks

We assume the reader is familiar with standard graph terminology, as contained for instance in [6]. Given a rooted tree, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v). Given a digraph $G = (V, E)$, and a set of vertices $S \subseteq V$, we denote by $G[S]$ the subgraph induced by S . We introduce next some of the building blocks of our new incremental algorithm.

2.1 Flow graphs, dominators, and bridges

A *flow graph* is a digraph with a distinguished *start vertex* s such that every vertex is reachable from s . Let $G = (V, E)$ be a strongly connected graph. The *reverse digraph* of G , denoted by $G^R = (V, E^R)$, is obtained by reversing the direction of all edges. Let s be a fixed but arbitrary start vertex of a strongly connected digraph G . Since G is strongly connected, all vertices are reachable from s and reach s , so we can view both G and G^R as flow graphs with start vertex s . To avoid ambiguities, throughout the paper we will denote those flow graphs respectively by G_s and G_s^R . Vertex u is a *dominator* of vertex v (u *dominates* v) in G_s if every path from s to v in G_s contains u . We let $Dom(v)$ denote be the set of dominators of v . The dominator relation can be represented by a tree D rooted at s , the *dominator tree* of G_s : u dominates v if and only if u is an ancestor of v in D . For any $v \neq s$, we denote by $d(v)$ the parent of v in D . Similarly, we

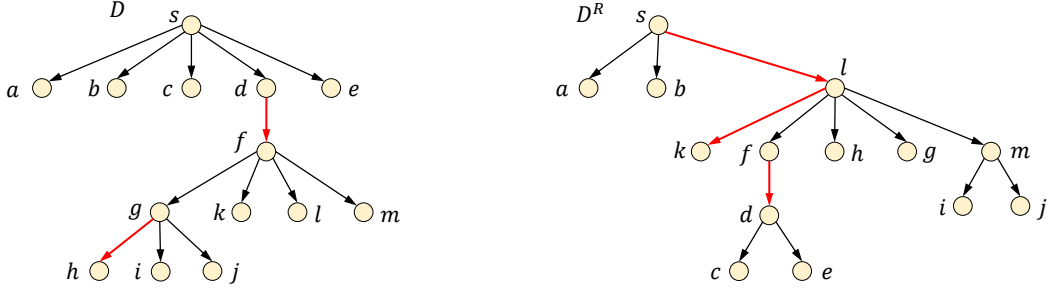


Figure 2: The dominator trees of flow graphs G_s and G_s^R . Strong bridges of G are shown red. (Better viewed in color.)

can define the dominator relation in the flow graph G_s^R , and let D^R denote the dominator tree of G_s^R , and $d^R(v)$ the parent of v in D^R . Lengauer and Tarjan [30] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [38]. Subsequently, several linear-time algorithms were discovered [2, 5, 10, 13]. An edge (u, v) is a *bridge* of a flow graph G_s if all paths from s to v include (u, v) .¹ Let s be an arbitrary start vertex of G . The following properties were proved in [26].

Property 2.1. ([26]) *Let s be an arbitrary start vertex of G . An edge $e = (u, v)$ is strong bridge of G if and only if it is a bridge of G_s (so $u = d(v)$) or a bridge of G_s^R (so $v = d^R(u)$) or both.*

As a consequence, of Property 2.1, all the strong bridges of the digraph G can be obtained from the bridges of the flow graphs G_s and G_s^R , and thus there can be at most $2(n - 1)$ strong bridges overall. Figure 2 illustrates the dominator trees D and D^R of the flow graphs G_s and G_s^R that correspond to the strongly connected digraph G of Figure 1. After deleting from the dominator trees D and D^R respectively the bridges of G_s and G_s^R , we obtain the *bridge decomposition* of D and D^R into forests \mathcal{D} and \mathcal{D}^R . Throughout the paper, we denote by D_u (resp., D_u^R) the tree in \mathcal{D} (resp., \mathcal{D}^R) containing vertex u , and by r_u (resp., r_u^R) the root of D_u (resp., D_u^R). The following lemma from [14] holds for a flow graph G_s of a strongly connected digraph G (and hence also for the flow graph G_s^R of G^R).

Lemma 2.2. ([14]) *Let G be a strongly connected digraph and let (u, v) be a strong bridge of G . Also, let D be the dominator tree of the flow graph G_s , for an arbitrary start vertex s . Suppose $u = d(v)$. Let w be any vertex that is not a descendant of v in D . Then there is path from w to v in G that does not contain any proper descendant of v in D . Moreover, all simple paths in G from w to any descendant of v in D must contain the edge $(d(v), v)$.*

2.2 Loop nesting forests and bridge-dominated components

Let G be a digraph. A *loop nesting forest* represents a hierarchy of strongly connected subgraphs of G [39], defined with respect to a dfs tree T of G , as follows. For any vertex u , the *loop* of u , denoted by $loop(u)$ is the set of all descendants x of u in T such that there is a path from x to u in G containing only descendants of u in T . Any two vertices in $loop(u)$ reach each other. Therefore,

¹Throughout the paper, to avoid confusion we use consistently the term *bridge* to refer to a bridge of a flow graph and the term *strong bridge* to refer to a strong bridge in the original graph.

$\text{loop}(u)$ induces a strongly connected subgraph of G ; it is the unique maximal set of descendants of u in T that does so. The $\text{loop}(u)$ sets form a laminar family of subsets of V : for any two vertices u and v , $\text{loop}(u)$ and $\text{loop}(v)$ are either disjoint or nested. The *loop nesting forest* H of G , with respect to T , is the forest in which the parent of any vertex v , denoted by $h(v)$, is the nearest proper ancestor u of v in T such that $v \in \text{loop}(u)$ if there is such a vertex u , and null otherwise. Then $\text{loop}(u)$ is the set of all descendants of vertex u in H , which we will also denote as $H(u)$ (the subtree of H rooted at vertex u). A loop nesting forest can be computed in linear time [5, 39]. Since we deal with strongly connected digraphs, each vertex is contained in a loop, so H is a tree. Therefore, we will refer to H as the *loop nesting tree* of G . Let $e = (u, v)$ be a bridge of the flow graph G_s , and let $G[D(v)]$ denote the subgraph induced by the vertices in $D(v)$. Let C be an SCC of $G[D(v)]$: we say that C is an *e-dominated component* of G . We also say that $C \subseteq V$ is a *bridge-dominated component* if it is an e -dominated component for some bridge e : As shown in the following lemma, bridge-dominated components form a laminar family, i.e., any two components in \mathcal{C} are either disjoint or one contains the other.

Lemma 2.3. *Let \mathcal{C} be the set family of all bridge-dominated components. Then \mathcal{C} is laminar.*

Proof. Let C and C' be two different sets of \mathcal{C} such that $C \cap C' \neq \emptyset$. Let e and e' be the nearest bridge ancestors of C and C' , respectively, such that C is e -dominated and C' is e' -dominated. Since C and C' contain a common vertex, say v , we can assume, without loss of generality, that e is an ancestor of e' . Any two vertices, $w \in C$ and $z \in C'$, are strongly connected, since they are both strongly connected with v . Moreover, both w and z are descendants of e , and hence it must be $C' \subset C$. This implies that any two components in \mathcal{C} are either disjoint or one contains the other, which yields the lemma. \square

Let $e = (u, v)$ be a bridge of G_s , and let w be a vertex in $D(v)$ such that $h(w) \notin D(v)$. As shown in [17], $H(w)$ induces an SCC in $G[D(v)]$, and thus it is an e -dominated component.

2.3 Bridge decomposition and auxiliary graphs

Now we define a notion of *auxiliary graphs* that play a key role in our approach. Auxiliary graphs were defined in [14] to decompose the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original 2-edge-connected blocks of G . Unfortunately, the auxiliary graphs of [14] are not suitable for our purposes, and we need a slightly different definition. For each root r of a tree in the bridge decomposition \mathcal{D} we define the *auxiliary graph* $\hat{G}_r = (V_r, E_r)$ of r as follows. The vertex set V_r of \hat{G}_r consists of all the vertices in D_r . The edge set E_r contains all the edges of G among the vertices of V_r , referred to as *ordinary* edges, and a set of *auxiliary* vertices, which are obtained by contracting vertices in $V \setminus V_r$, as follows. Let v be a vertex in V_r that has a child w in $V \setminus V_r$. Note that w is a root in the bridge decomposition \mathcal{D} of D . For each such child w of v , we contract w and all its descendants in D into v . Figure 3 shows the bridge decomposition of the dominator tree D and the corresponding auxiliary graphs. The bridge decomposition of D^R is shown in Figure 4. Differently from [14], our auxiliary graphs do not preserve the 2-edge-connected blocks of G . Note that each vertex appears exactly in one auxiliary graph. Furthermore, each original edge corresponds to at most one auxiliary edge. Therefore, the total number of vertices in all auxiliary graphs is n , and the total number of edges is at most m . We use the term *auxiliary components* to refer to the SCCs of the auxiliary graphs.

Lemma 2.4. *All the auxiliary graphs of a flow graph G_s can be computed in linear time.*

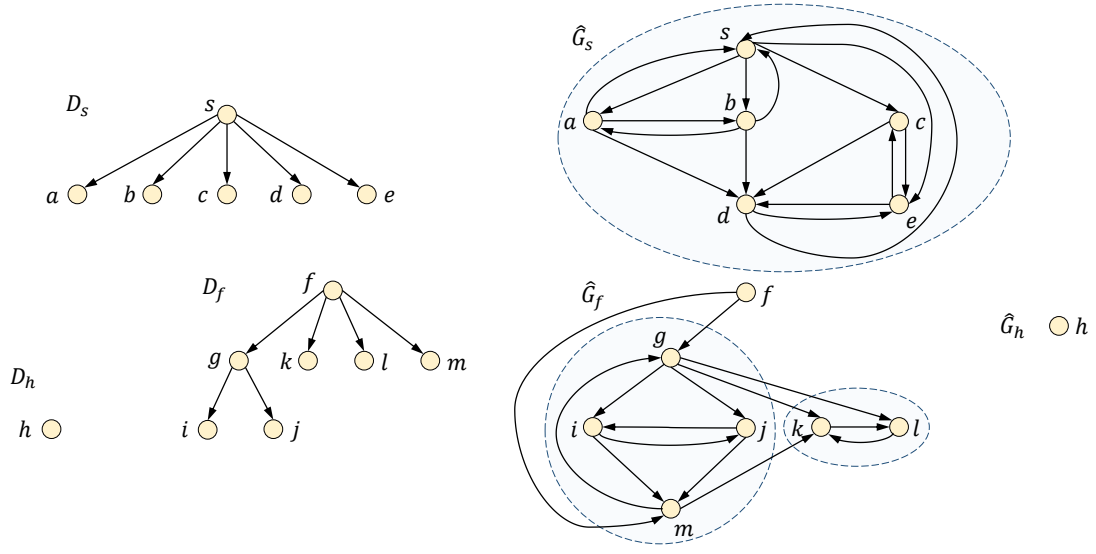


Figure 3: The bridge decomposition of the dominator tree D of Figure 2, the corresponding auxiliary graphs \hat{G}_r and their SCCs shown encircled.

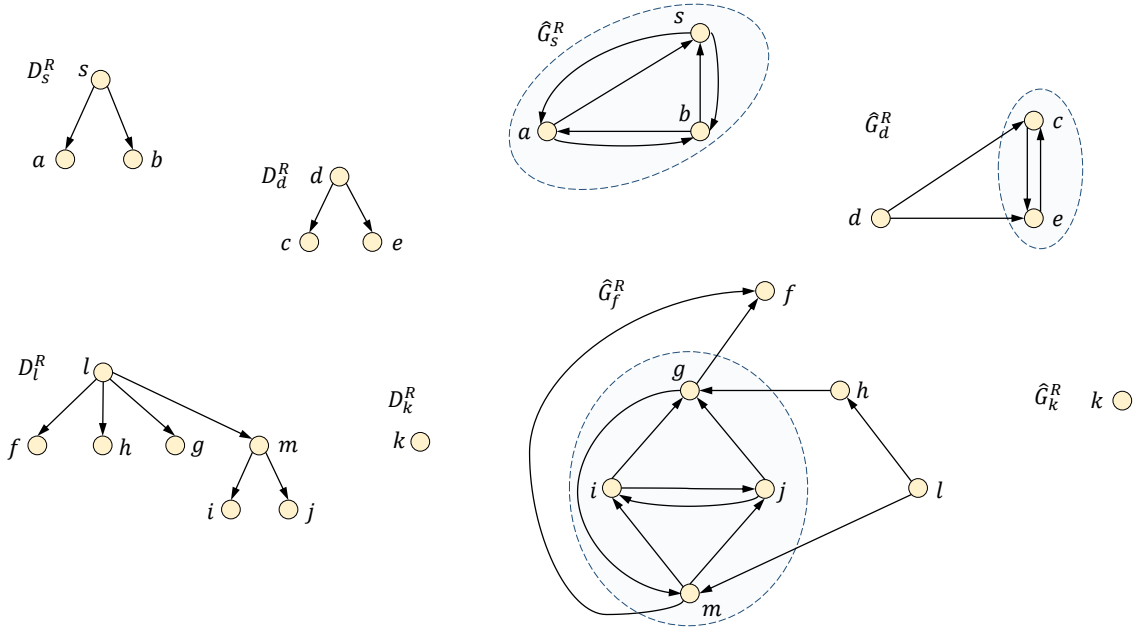


Figure 4: The bridge decomposition of the dominator tree D^R of Figure 2, the corresponding auxiliary graphs \hat{G}_r^R and their SCCs shown encircled.

Proof. To construct the auxiliary graph $\widehat{G}_r = (V_r, E_r)$ we need to specify how to compute the shortcut edge of each edge entering V_r from a descendant of r in $V \setminus V_r$. To do this efficiently we need to test ancestor-descendant relations in D . We can test this relation in constant time using a preorder numbering of D [37]. Let $pre(v)$ denote the preorder number of a vertex v in D . Suppose (u, v) is an edge such that $v \in V_r$, $u \notin V_r$, and u is a descendant of r in D . We need to find the nearest ancestor u' of u in D such that $u' \in V_r$. Then, (u, v) corresponds to the shortcut edge (u', v) . To compute the shortcut edges of \widehat{G}_r , we create a list B_r that contains the edges (u, v) such that $u \in V_r$, $u \notin V_r$, and u is a descendant of r in D . For each such edge (u, v) we need to find the nearest ancestor u' of u in D such that $u' \in V_r$. Then, (u, v) corresponds to the shortcut edge (u', v) . We create a second list B'_r that contains the vertices in V_r that have a child that is not in V_r , and sort B'_r in increasing preorder. Then u' is the last vertex in the sorted list B'_r such that $pre(u') \leq pre(u)$. Thus the shortcut edges can be computed by bucket sorting and merging. In order to do these computations in linear time for all auxiliary graphs, we sort all the lists at the same time as follows. First, we create a unified list B containing the triples $\langle r, pre(u), v \rangle$ for each edge (u, v) that corresponds to a shortcut edge in the auxiliary graph \widehat{G}_r . Next we sort B in increasing order of the first two elements. We also create a second list B' with pairs $\langle r, pre(u') \rangle$, where u' is a vertex in V_r that has a child that is not in V_r , and sort the pairs in increasing order. Finally, we compute the shortcut edges of each auxiliary graph \widehat{G}_r by merging the sorted sublists of B and B' that correspond to the same root r . Then, the shortcut edge for the triple $\langle r, pre(u), v \rangle$ is (u', v) , where $\langle r, pre(u') \rangle$ is the last pair in the sorted sublist of B' with root r such that $pre(u') \leq pre(u)$. \square

Let C be a set of vertices, and let (u, v) be a bridge of G_s . The restriction of C in D_v is the set $C_v = C \cap D_v$.

Lemma 2.5. *Let $e = (u, v)$ be a bridge of G_s , and let C be an e -dominated component. Then, the restriction C_v of C in D_v is an SCC of \widehat{G}_v .*

Proof. Let x and y be any vertices in D_v . It suffices to argue that x and y are in the same e -dominated component if and only if they are strongly connected in \widehat{G}_v . First suppose that x and y are in different e -dominated components. Then x and y are not strongly connected in the subgraph $G[D(v)]$ that is induced by $D(v)$. Then, without loss of generality, we can assume that all paths from x to y contain a vertex in $V \setminus D(v)$. By Lemma 2.2 all paths from x to y contain (u, v) . Hence x and y are not strongly connected in \widehat{G}_v .

Now suppose that x and y are in the same e -dominated component. That is, x and y are strongly connected in $G[D(v)]$, so there is a path from x to y and a path from y to x containing only vertices in $D(v)$. Then \widehat{G}_v contains, by construction, a path from x to y and a path from y to x . Thus x and y are strongly connected in \widehat{G}_v . \square

2.4 A new algorithm for 2-edge-connected blocks

We next sketch a new linear-time algorithm to compute the 2-edge-connected blocks of a strongly connected digraph G that combines ideas from [14] and [17] and that will be useful for our incremental algorithm. We refer to this algorithm as the **2ECB labeling algorithm**. Similarly to the algorithm of [17], our algorithm assigns a label to each vertex, so that two vertices are 2-edge-connected if and only if they have the same label. The labels are defined by the bridge decomposition of the dominator trees and by the auxiliary components, as follows. Let \widehat{G}_r be an auxiliary graph of G_s . We pick a *canonical vertex* for each SCC C of \widehat{G}_r , and denote by c_x the canonical vertex of the SCC that contains x . We define c_x^R for the SCC's of the auxiliary graphs of G_s^R analogously. We define the label of x as $label(x) = \langle r_x, c_x, r_x^R, c_x^R \rangle$.

To prove that the 2ECB labeling algorithm is correct, we show that the labels produced by the algorithm are essentially identical to the labels of [17]. In order to do that, we briefly review the linear-time algorithm of [17] for computing the 2-edge-connected blocks of G . The algorithm of [17] computes, for each vertex x , a tuple $label'(x) = \langle r_x, h_x, r_x^R, h_x^R \rangle$, where r_x and r_x^R are exactly as before, while h_x and h_x^R are defined by the loop nesting trees H and H^R respectively. We say that a vertex x is a *boundary vertex* in H if $h(x) \notin D_x$, i.e., when x and its parent in H lie in different trees of the bridge decomposition \mathcal{D} . As a special case, we also let s be a boundary vertex of H . The *nearest boundary vertex* of x in H , denoted by h_x , is the nearest ancestor of x in H that is a boundary vertex in H . Hence, if $r_x = s$ then $h_x = s$. Otherwise, h_x is the unique ancestor of x in H such that $h_x \in D_x$ and $h(h_x) \notin D_x$. We define the *nearest boundary vertex* of x in H^R similarly. A vertex x is a *boundary vertex* in H^R if $h^R(x) \notin D_x^R$, i.e., when x and its parent in H^R lie in different trees of \mathcal{D}^R . Again, we let s be a boundary vertex of H^R . Then, the *nearest boundary vertex* of x in H^R , denoted by h_x^R , is the nearest ancestor of x in H^R that is a boundary vertex in H^R . As shown in [17], two vertices x and y are 2-edge-connected if and only if $label'(x) = label'(y)$.

Lemma 2.6. *Let x and y be any vertices of G . Then, x and y are 2-edge-connected if and only if $label(x) = label(y)$.*

Proof. Let $label'(v) = \langle r_v, h_v, r_v^R, h_v^R \rangle$ be the labels assigned to by the original labeling algorithm of [17]. By [17], we have that x and y are 2-edge-connected if and only if $label'(x) = label'(y)$. Suppose $r_x = r_y$. We show that $h_x = h_y$ if and only if $c_x = c_y$. The fact that $r_x = r_y$ implies that x and y are in the same auxiliary graph \widehat{G}_{r_x} . By Lemma 2.5, x and y are strongly connected in \widehat{G}_{r_x} if and only if $h_x = h_y$. Hence, $h_x = h_y$ if and only if $c_x = c_y$. The same argument implies that if $r_x^R = r_y^R$, then $h_x^R = h_y^R$ if and only if $c_x^R = c_y^R$. We conclude that x and y are 2-edge-connected if and only if $label(x) = label(y)$. \square

Theorem 2.7. *The 2ECB labeling algorithm computes the 2-edge-connected blocks of a strongly connected digraph in linear time.*

Proof. The correctness of the labeling algorithm follows from Lemma 2.6. We now bound the running time. The dominator trees and the bridges can be computed in linear time [5]. Also all the auxiliary graphs and the auxiliary components can be computed in linear time by Lemma 2.4 and [36]. Hence, all the required labels can be computed in linear time. \square

2.5 Incremental dominators and incremental SCCs

We will use two other building blocks for our new algorithm, namely incremental algorithms for maintaining dominator trees and SCCs. As shown in [16], the dominator tree of a flow graph with n vertices can be maintained in $O(m \min\{n, k\} + kn)$ time during a sequence of k edge insertions, where m is the total number of edges after all insertions. For maintaining the SCCs of a digraph incrementally, Bender et al. [4] presented an algorithm that can handle the insertion of m edges in a digraph with n vertices in $O(m \min\{m^{1/2}, n^{2/3}\})$ time. Since we aim at an $O(mn)$ bound, we can maintain the SCCs with a simpler data structure based on topological sorting [32], augmented so as to handle cycle contractions, as suggested by [19]. We refer to this data structure as *IncSCC*, and we will use it both for maintaining the SCCs of the input graph, and the auxiliary components (i.e., the SCCs of the auxiliary graphs). We maintain the SCCs and a topological order for them. Each SCC is represented by a canonical vertex, and the partition of the vertices into SCCs is maintained through a disjoint set union data structure [38, 40]. The data structure supports the operation *unite*(p, q), which, given canonical vertices p and q , merges the SCCs containing p and q into one

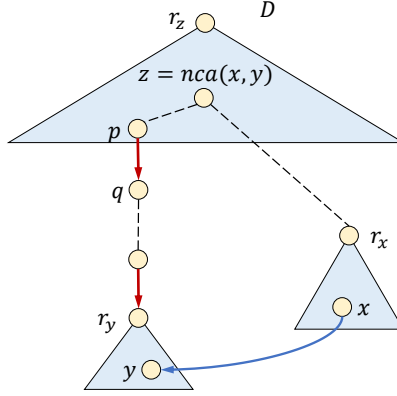


Figure 5: The bridge decomposition of D before the insertion of a new edge (x, y) .

new SCC and makes p the canonical vertex of the new SCC. It also supports the query $find(v)$, which returns the canonical vertex of the SCC containing v . Here we use the abbreviation $f(v)$ to stand for $find(v)$. The topological order is represented by a simple numbering scheme, where each canonical vertex is numbered with an integer in the range $[1, n]$, so that if (u, v) is an edge of G , then either $f(u) = f(v)$ (u and v are in the same SCC) or $f(u)$ is numbered less than $f(v)$ (when u and v are in different SCCs). With each canonical vertex p we store a list $out(p)$ of edges leaving vertices that are in the same SCC as p , i.e., edges (u, v) with $f(u) = p$. Note that $out(p)$ may contain multiple vertices in the same SCC (i.e., vertices u and v with $f(u) = f(v)$), due to the SCC contractions (and shortcut edges, in case of the auxiliary components) during edge insertions. Also, $out(p)$ may contain loops, that is, vertices v with $f(v) = p$. Each out list is stored as a doubly linked circular list, so that we can merge two lists and delete a vertex from a list in $O(1)$. When the incremental SCC data structure detects that a new SCC is formed, it locates the SCCs that are merged and chooses a canonical vertex for the new SCC. The IncSCC data structure can handle m edge insertions in a total of $O(mn)$ time.

3 Incremental 2-edge-connectivity in strongly connected digraphs

To maintain the 2-edge-connected blocks of a strongly connected digraph during edge insertions, we design an incremental version of the labeling algorithm of Section 2.4. If labels are maintained explicitly, one can answer in $O(1)$ time queries on whether two vertices are 2-edge-connected, and report in $O(n)$ time all the 2-edge-connected blocks (see Section 5). Let (x, y) be the edge to be inserted. We say that vertex v is *affected* by the update if $d(v)$ (its parent in D) changes. Note that $Dom(v)$ may change even if v is not affected. Similarly, an auxiliary component (resp., auxiliary graph) is affected if it contains an affected vertex. We let $nca(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D . We also denote by $D[u, v]$ the path from vertex u to vertex v in D . If $nca(x, y)$ and y are in different subtrees in the bridge decomposition of D before the insertion of the edge (x, y) , we let (p, q) be the first bridge encountered on the path $D[nca(x, y), y]$ (Figure 5). For any vertex v , we denote by $depth(v)$ the depth of v in D .

3.1 Affected vertices and canceled bridges

There are affected vertices after the insertion of (x, y) if and only if $nca(x, y)$ is not a descendant of $d(y)$ [35]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [3].

Lemma 3.1. ([16]) *A vertex v is affected after the insertion of edge (x, y) if and only if $\text{depth}(nca(x, y)) < \text{depth}(d(v))$ and there is a path π in G from y to v such that $\text{depth}(d(v)) < \text{depth}(w)$ for all $w \in \pi$. If v is affected, then it becomes a child of $nca(x, y)$ in D .*

The algorithm in [16] applies Lemma 3.1 to identify the affected vertices by starting a search from y (if y is not affected, then no other vertex is). We assume that the outgoing and incoming edges of each vertex are maintained as linked lists, so that a new edge can be inserted in $O(1)$, and that the dominator tree D is represented by the parent function d . We also maintain the depth of vertices in D . We say that a vertex v is *scanned*, if the edges leaving v are examined during the search for affected vertices, and that it is *visited* if there is a scanned vertex u such that (u, v) is an edge in G . Every scanned vertex is either affected or a descendant of an affected vertex in D . By Lemma 3.1, a visited vertex v is scanned if $\text{depth}(nca(x, y)) < \text{depth}(d(v))$. Let (u, v) be a bridge of G_s . We say that (u, v) is *canceled* by the insertion of edge (x, y) if it is no longer a bridge after the insertion. We say that a bridge (u, v) is *locally canceled* if (u, v) is a canceled bridge and v is not affected. Note that if (u, v) is locally canceled, then $u = nca(x, y)$. In the next lemmata, we consider the effect of the insertion of edge (x, y) on the bridges of G_s , and relate the affected and scanned vertices with the auxiliary components. Recall that (p, q) is the first bridge encountered on the path $D[nca(x, y), y]$ (Figure 5), $D(v)$ denotes the descendants of v in D , and $G[C]$ is the subgraph induced by the vertices in C .

Lemma 3.2. *Suppose that bridge (p, q) is not locally canceled after the insertion of (x, y) . Let $z = nca(x, y)$ and let v be an affected vertex such that $r_v \neq r_z$. All vertices reachable from v in $G[D(q)]$ are either affected or scanned.*

Proof. Let π be a path in $G[D(q)]$ from v to another vertex w . Since (p, q) is not locally canceled we have that $z \neq p$ or $w \neq q$. Hence, all vertices u on π satisfy $\text{depth}(z) < \text{depth}(d(u))$. We prove the lemma by induction on the number of the edges in π that are not edges of the dominator tree D . Suppose that π consists of only one edge, i.e., $\pi = (v, w)$. If w is a child of v in D , then w is scanned. Otherwise, the parent property of D [18] implies that $d(w)$ is an ancestor of v . In this case, Lemma 3.1 implies that w is affected. Thus, the induction base holds. Assume by induction that the lemma holds for any vertex that is reachable in $G[D(q)]$ from an affected vertex through a path of at most k edges that are not edges of D . Let π be a path from v to w . Let (u, w') be the first edge on π such that w' is not a descendant of v in D . The parent property of D implies again that $d(w')$ is an ancestor of u in D . Since w' is not a descendant of v , $d(w')$ is also an ancestor of v in D . Hence, Lemma 3.1 implies that w' is affected. The part of π from w' to w satisfies the induction hypothesis, and the lemma follows. \square

Lemma 3.3. *Let $e = (u, v)$ be a bridge of G_s that is canceled by the insertion of edge (x, y) . Then (i) y is a descendant of v in D , and (ii) y is in the same e -dominated component as v .*

Proof. Since $e = (u, v)$ is not a bridge in G'_s , there must be a path π from s to v in G'_s that avoids e . This path does not exist in G_s , so π contains (x, y) . Consider the subpath π_1 of π from y to v . Path π_1 exists in G_s and avoids e . So, Lemma 2.2 implies that all vertices in π_1 (y included) are descendants of v in D , since otherwise π_1 would have to include e . \square

Corollary 3.4. *A bridge $e = (u, v)$ of G_s is canceled by the insertion of edge (x, y) if and only if $\text{depth}(\text{nca}(x, y)) \leq \text{depth}(u)$ and there is a path π in G from y to v such that $\text{depth}(u) < \text{depth}(w)$ for all $w \in \pi$.*

By Corollary 3.4, we can use the incremental algorithm of [16] to detect canceled bridges, without affecting the $O(mn)$ bound. Indeed, suppose $e = (u, v)$ is a canceled bridge. By Lemma 3.3, y is a descendant of v in D and in the same e -dominated component as v . Hence, v will be visited by the search from y .

If a bridge (u, v) is locally canceled, there can be vertices in D_v that are not scanned, and that after the insertion will be located in D_u , without having their depth changed. This is a difficult case for our analysis: fortunately, the following lemma shows that this case can happen only $O(n)$ times overall.

Lemma 3.5. *Suppose (u, v) is a bridge of G_s that is locally canceled by the insertion of edge (x, y) . Then (u, v) is no longer a strong bridge in G after the insertion.*

Proof. Since v is visited but not affected, we have that $u = \text{nca}(x, y)$. Then x is a descendant of u in D , and $y \notin D[u, x]$. Hence, there is a path π_1 from u to x that does not contain y . Since y is affected, Lemma 3.3 implies that there is a path π_2 from y to v that contains only descendants of v . So, $\pi_1 \cdot (x, y) \cdot \pi_2$ is a path from u to v that avoids (u, v) in G after the insertion of (x, y) . \square

Note that a canceled bridge that is not locally canceled may still appear as a bridge in G_s^R after the insertion of edge (x, y) . Next we provide some lemmata that help us to identify the necessary changes in the auxiliary components of the affected subgraphs and \hat{G}_{r_z} .

Lemma 3.6. *Let v be a vertex that is affected by the insertion of edge (x, y) . Then r_v is on the path $D[r_z, r_y]$.*

Proof. Vertices x and y are descendants of z , so r_y and r_x are descendants of r_z . Since v is affected, v is a descendant of z , and by Lemma 3.1 there is a path π from y to v such that $\text{depth}(d(v)) < \text{depth}(w)$ for all $w \in \pi$. Thus, π does not contain the bridge $(d(r_v), r_v)$, so by Lemma 2.2, y is a descendant of r_v . Then r_v is a descendant of r_z and an ancestor of r_y . \square

Lemma 3.7. ([18]) *Let S be the set of vertices of a strongly connected subgraph of G . Then S consists of a set of siblings in D and possibly some of their descendants in D .*

In the following, we assume that bridge (p, q) is not locally canceled after the insertion of (x, y) and that $z = \text{nca}(x, y)$.

Lemma 3.8. *Let C be an affected auxiliary component of an auxiliary graph \hat{G}_r with $r \neq r_z$. Then C consists of a set of affected siblings in D and possibly some of their affected or scanned descendants in D .*

Proof. By Lemma 3.7, C consists of a set of siblings S in D and possibly some of their descendants in D . Also, Lemma 3.2 implies that all vertices in C are scanned. So, it suffices to show that all siblings in S are affected. Let v be an affected vertex in C . Consider any sibling $u \in S$. Since C is strongly connected, there is a path π_2 from v to u containing only vertices in C . Since (p, q) is not locally canceled, $u \neq q$ or $z \neq p$, so all vertices w on π_2 satisfy $\text{depth}(z) < \text{depth}(d(w))$. Let π_1 a path from y to v that satisfies Lemma 3.1. Then $\pi_1 \cdot \pi_2$ is a path from y to u that also satisfies Lemma 3.1. Hence u is affected. \square

An auxiliary component is *scanned* if it contains a scanned vertex. As with vertices, affected auxiliary components are also scanned (the converse is not necessarily true).

Lemma 3.9. *Let C be a scanned auxiliary component of an auxiliary graph \widehat{G}_r with $r \neq r_z$. Then all vertices in C are scanned.*

Proof. The fact that (p, q) is not locally canceled implies that $q \notin C$ or $p \neq z$. So, for each vertex w in C we have $\text{depth}(z) < \text{depth}(d(w))$, which implies that w is scanned. \square

We say that a vertex v is *moved* if it is located in an auxiliary graph \widehat{G}_r with $r \neq r_z$ before the insertion of (x, y) , and in \widehat{G}_{r_z} after the insertion. Lemmata 3.8 and 3.9 imply that if an auxiliary component C contains a moved vertex, then all vertices in the component are also moved. We call such an auxiliary component *moved*. Now we describe how to find the moved auxiliary components that need to be merged. Let H be the subgraph of G induced by the scanned vertices in $D(q)$. We refer to H as the *scanned subgraph*.

Lemma 3.10. *Let ζ and ξ be two distinct roots in the bridge decomposition of D , such that $\zeta, \xi \neq r_z$, and D_ζ and D_ξ are contained in $D(q)$. Let C_ζ and C_ξ be scanned components in \widehat{G}_ζ and \widehat{G}_ξ , respectively. Then C_ζ and C_ξ are strongly connected in $G[D(q)]$ if and only if they are strongly connected in H .*

Proof. Clearly, C_ζ and C_ξ are strongly connected in $G[D(q)]$ if they are strongly connected in H , so it remains to prove the converse. Suppose C_ζ and C_ξ are strongly connected in $G[D(q)]$. Then, there is a path π in $G[D(q)]$ from a vertex in C_ξ to a vertex in C_ζ . The fact that (p, q) is not locally canceled implies that $q \notin C_\zeta \cup C_\xi$ or $z \neq p$. Then, for each vertex w on π we have $\text{depth}(z) < \text{depth}(d(w))$, which implies that w is scanned. Hence π exists in H . Similarly, there is a path in $G[D(q)]$ from a vertex in C_ζ to a vertex in C_ξ that is also contained in H . \square

Now we introduce a dummy root r^* in H , together with an edge (v, r^*) for each scanned vertex v that has a leaving edge (v, w) such that $w \in D_z$ and w is in the auxiliary component of p in \widehat{G}_{r_z} . We denote this graph by H^* .

Lemma 3.11. *A scanned vertex $v \notin D_z$ is strongly connected in $G[D(r_z)]$ to a vertex $w \in D_z$ if and only if r^* is reachable from v in H^* . In this case, v and p are also strongly connected in $G[D(r_z)]$.*

Proof. Let $v \notin D_z$ be a scanned vertex. By Lemma 3.6, v is in $D(q)$, hence a descendant of p in D . Suppose r^* is reachable from v in H^* . Then, there is a path π_1 in $G[D(r_z)]$ from v to a vertex $w \in D_z$, where w is in the auxiliary component of p in \widehat{G}_{r_z} . Since p and w are in the same auxiliary component, there is a path π_2 from w to p in $G[D(r_z)]$. Also, since v is a descendant of p in D , there is a path π_3 from p to v in $G[D(r_z)]$. Paths π_1 and $\pi_2 \cdot \pi_3$ imply that w and v are strongly connected in $G[D(r_z)]$.

Conversely, let w be a vertex in D_z that is strongly connected in $G[D(r_z)]$ to a scanned vertex $v \notin D_z$. Let π_1 be a path from w to v in $G[D(r_z)]$, and let π_2 be a path from v to w in $G[D(r_z)]$. From Lemma 3.2 we have that $v \in D(q)$, hence Lemma 2.2 implies that π_1 contains (p, q) so $p \in \pi_1$. Thus p and w are also strongly connected in $G[D(r_z)]$. Now let w' be the first vertex on π_2 that is in D_z , and let (t, w') be the edge in π_2 that enters w' . Then w' and w are also strongly connected in $G[D(r_z)]$. Also, since (p, q) is not locally canceled, we have $q \notin \pi_2$ or $z \neq p$. Then, for each vertex t' on the part of π_2 from v to t we have $\text{depth}(z) < \text{depth}(d(t'))$, which implies that t' is scanned. Hence π_2 exists in H , so by construction, v reaches r^* in H^* . \square

3.2 The Algorithm

We describe next our incremental algorithm for maintaining the 2-edge-connected blocks of a strongly connected digraph G . We refer to this algorithm as $\text{SCInc2ECB}(G)$. We initialize the algorithm

and the associated data structures by executing the labeling algorithm of Section 2.4. Algorithm `Initialize(G, s)`, shown below, computes the dominator tree D , the set of bridges Br of flow graph G_s , the bridge decomposition \mathcal{D} of D , and the corresponding auxiliary graphs \hat{G}_r . Finally, for each auxiliary graph \hat{G}_r , it finds its auxiliary components, computes the labels r_w and c_w for each vertex $w \in V_r$, and initializes an `IncSCC` data structure. The execution of `Initialize(G^R, s)` performs analogous steps in the reverse flow graph G_s^R .

Algorithm 1: `Initialize(G, s)`

```

1 Set  $s$  to be the designated start vertex of  $G$ .
2 Compute the dominator tree  $D$  and the set of bridges  $Br$  of the corresponding flow graph  $G_s$ .
3 Compute the bridge decomposition  $\mathcal{D}$  of  $D$ .
4 foreach root  $r$  in  $\mathcal{D}$  do
5   Compute the auxiliary graph  $\hat{G}_r$  of  $r$ .
6   Compute the strongly connected components in  $\hat{G}_r$ .
7   foreach strongly connected component  $C$  in  $\hat{G}_r$  do
8     Choose a vertex  $v \in C$  as the canonical vertex of the auxiliary component  $C$ .
9     foreach vertex  $w \in C$  do
10      Set  $r_w = r$  and  $c_w = v$ .
11    end
12  end
13  Initialize a IncSCC data structure for  $\hat{G}_r$ .
14 end
```

Algorithm 2: `SCInsertEdge(G, e)`

```

1 Let  $s$  be the designated start vertex of  $G$ , and let  $e = (x, y)$ .
2 Compute the nearest common ancestor  $z$  and  $z^R$  of  $x$  and  $y$  in  $D$  and  $D^R$  respectively.
3 Update the dominator trees  $D$  and  $D^R$ , and return the lists  $S$  and  $S^R$  of the vertices that
  were scanned in  $D$  and  $D^R$  respectively.
4 if a bridge is locally canceled in  $G_s$  or in  $G_s^R$  then
5   Execute Initialize( $G, s$ ) and Initialize( $G^R, s$ ).
6 else
7   Execute UpdateAC( $\mathcal{D}, z, x, y, S$ ) and UpdateAC( $\mathcal{D}^R, z^R, y, x, S^R$ ).
8 end
```

When a new edge $e = (x, y)$ is inserted, algorithm `SCInc2ECB` executes procedure `SCInsertEdge(G, e)`, which updates dominator trees D and D^R , together with the corresponding bridge decompositions. It also finds the set of scanned vertices in G_s and G_s^R . If a bridge of D or D^R is locally cancelled, then we restart the algorithm by executing `Initialize`. Otherwise, we need to update the auxiliary components in G_s and G_s^R . These updates are handled by procedure `UpdateAC`. Before describing `UpdateAC`, we provide some details on the implementation of the `IncSCC` data structures, which maintain the auxiliary components of each auxiliary graph \hat{G}_r using the “one-way search” structure of [19, Sections 2 and 6]. Since we need to insert and delete canonical vertices, we augment this data structure as follows. We maintain the canonical vertices of each auxiliary component in a linked list L_r , arranged according to the given topological order of \hat{G}_r . For each vertex v in L_r , we also maintain a rank in L_r which is an integer in $[1, n]$ such that for any two canonical vertices u and v

Algorithm 3: UpdateAC(\mathcal{D}, z, x, y, L)

- 1 Let r_z be root of the tree D_z in \mathcal{D} that contains z .
 - 2 Let $c_{x'}$ be the canonical vertex of the nearest ancestor x' of x in D such that $x' \in D_z$.
 - 3 Let (p, q) be the first bridge on the path $D[z, y]$, and let c_p be the canonical vertex of p .
 - 4 Form the scanned graph H^* that contains the scanned vertices $S \setminus D_z$ and the edges among them.
 - 5 Compute the strongly connected components \mathcal{C} of $H^* \setminus r^*$ and order them topologically.
 - 6 Compute the components \mathcal{C}^* of \mathcal{C} that reach r^* in H^* .
 - 7 **foreach** *strongly connected component C in \mathcal{C}^* that is moved* **do**
 - 8 Merge C with the component of c_p .
 - 9 **end**
 - 10 **forall the** *strongly connected components in $\mathcal{C} \setminus \mathcal{C}^*$ that are moved* **do**
 - 11 Insert the components in the topological order of \widehat{G}_{r_z} just after the component of c_p .
 - 12 **end**
 - 13 **foreach** *vertex $w \in S$* **do**
 - 14 **if** w *is moved to \widehat{G}_{r_z}* **then** set $r_w = r_z$.
 - 15 **end**
 - 16 Update the lists of out edges in the IncSCC data structures of \widehat{G}_{r_z} and of the affected auxiliary graphs.
 - 17 Insert edge $(c_{x'}, y)$ in the list of outgoing edges of $c_{x'}$ and update the IncSCC data structure of \widehat{G}_{r_z} .
-

in L_r , $\text{rank}(u) < \text{rank}(v)$ if and only if u precedes v in L_r . The ranks of all vertices can be stored in a single array of size n . Also, with each canonical vertex w , we store a pointer to the location of w in L . We represent L_r with a doubly linked list so that we can insert and delete a canonical vertex in constant time. When we remove vertices from a list L_r we do not need to update the ranks of the remaining vertices in L_r . The insertion of an edge (x, y) may remove vertices from various lists L_r , but may insert vertices only in L_{r_z} . After these insertions, we recompute the ranks of all vertices in L_{r_z} just by traversing the list and assigning rank i to the i -th vertex in the list. We maintain links between an original edge e , stored in the adjacency lists of G , and at most one copy of e in a *out* list of IncSCC. This enables us to keep for each shortcut edge $e' = (v', w)$ a one-to-one correspondence with the original edge $e = (v, w)$ that created e' . We do that because if an ancestor of v is moved to the auxiliary graph \widehat{G}_{r_z} that contains v' ($v' = p$ in Figure 6), then e may correspond to a different shortcut edge or it may even become an ordinary edge of \widehat{G}_{r_z} . Using this mapping we can update the *out* lists of IncSCC. To initialize the IncSCC structure of an auxiliary graph, we compute a topological order of the auxiliary components in \widehat{G}_r , and create the list of outgoing edges $\text{out}(v)$ for each canonical vertex v .

If inserting edge (x, y) does not locally cancel a bridge in G_s and G_s^R , then we update the auxiliary components of G_s using procedure UpdateAC(\mathcal{D}, z, x, y, S), where \mathcal{D} is the updated bridge decomposition of D , $z = \text{nca}(x, y)$, and S is a list of the vertices scanned during the update of D . We do the same to update the auxiliary components of G_s^R . Procedure UpdateAC first computes the auxiliary components that are moved to \widehat{G}_{r_z} , possibly merging some of them, and then inserts the edge (x, y) as an original or a shortcut edge of \widehat{G}_{r_z} , depending on whether $x \in D_{r_z}$ or not. Note that the insertion of (x, y) may cause the creation of a new auxiliary component in \widehat{G}_{r_z} . Now we specify some further details in the implementation of UpdateAC. The vertices that are moved to \widehat{G}_{r_z} are the scanned vertices in S that are not descendants of a strong bridge. Hence, we can mark the

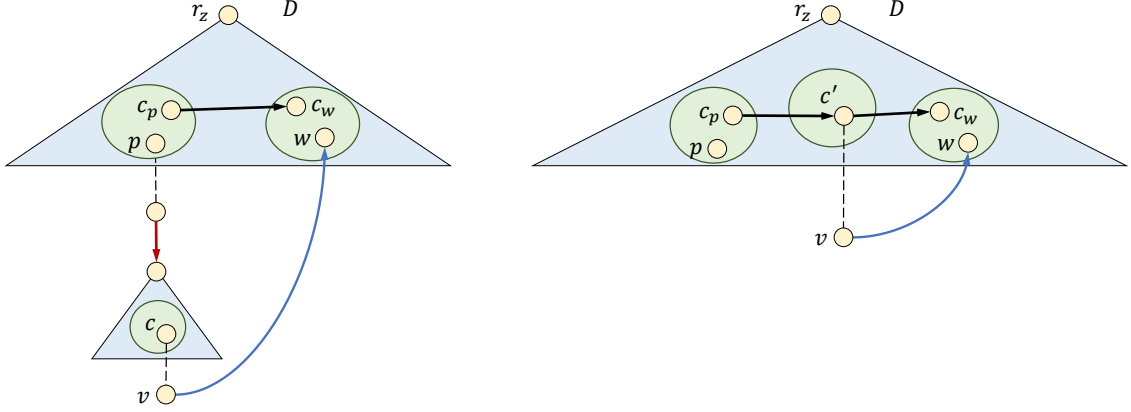


Figure 6: Before the insertion of (x, y) , edge (v, w) corresponds to the shortcut edge (p, w) of \hat{G}_{r_z} , and is stored in $out(c_p)$. An auxiliary component with canonical vertex c is affected by the insertion of (x, y) and is merged into a component with canonical vertex c' ($c' = c$ if the component is moved without merging with another component). Now c' becomes the canonical vertex of the nearest ancestor of v in D_z , and edge (v, w) is stored as a shortcut edge in $out(c')$.

vertices that are moved to \hat{G}_{r_z} during the search for affected vertices. The next task is to update the *out* lists of the canonical vertices in \hat{G}_{r_z} and the affected auxiliary graphs. We process the list of scanned vertices S as follows. Let v be such a vertex. If v is not marked, i.e., is not moved to \hat{G}_{r_z} , then we process the edges leaving v ; otherwise, we process both the edges leaving v and the edges entering v . Suppose v is marked. Let (v, w) be an edge leaving v in G . If w is also in \hat{G}_{r_z} after the insertion, then we add the edge (v, w) in $out(f(v))$. Moreover, if w is not in S , then it was already located in \hat{G}_{r_z} before the insertion, so we delete the shortcut edge stored in $out(f(p))$. If w is not in \hat{G}_{r_z} after the insertion, then (v, w) is a bridge in D and we do nothing. Now consider an edge (w, v) entering v in G . If w is scanned, then we will process (w, v) while processing the edges leaving w . Otherwise, w remains a descendant of p , so we insert the edge (w, v) in $out(f(p))$. Now we consider the unmarked scanned vertices v . Let (v, w) an edge leaving v in G . If $w \in D_z$, we insert the edge (v, w) into $out(f(v'))$, where v' is the nearest marked ancestor of v in D . Otherwise, if $w \notin D_z$, the edge (v'', w) , where v'' is the nearest ancestor of v in D_w , already exists since v was a descendant of v'' before the insertion of (x, y) . Next, we consider the updates in the L_r lists and the vertex ranks. While we process S , if we encounter a moved canonical vertex $v \in S$ that was located in an auxiliary graph \hat{G}_r with $r_z \neq r$, then we delete v from L_r . Note that we do not need to update the ranks of the remaining vertices in lists L_r with $r \neq r_z$. To update L_{r_z} , we insert the moved canonical vertices of the SCCs in $\mathcal{C} \setminus \mathcal{C}^*$, in a topological order of $H = H^* \setminus r^*$, just after $f(p)$. Then we traverse L_{r_z} and update the ranks of the canonical vertices. The final step is to actually insert edge (x, y) in the IncSCC data structure of \hat{G}_{r_z} . We do that by adding (x, y) in $out(f(x'))$, where x' is the nearest ancestor of x in D_z . If $rank(f(x')) > rank(f(y))$, then we execute the forward-search procedure of IncSCC.

Lemma 3.12. *Algorithm SCInc2ECB is correct.*

Proof. It suffices to show that UpdateAC correctly maintains: (i) the auxiliary components, and (ii) the IncSCC structure of each auxiliary graph. The fact that (i) holds follows from Lemmata 3.10 and 3.11.

To prove (ii), we need to show that the topological order and the *out* lists of each auxiliary graph are updated correctly in lines 10–12 and 16 of **UpdateAC**.

Suppose \widehat{G}_r is an auxiliary graph that is neither affected nor contains z . We argue that no shortcut edge in \widehat{G}_r needs to be replaced. By Lemma 3.6, we only need to consider edges (u, v) such that $v \in D_r$ and u is a descendant of an affected vertex. We distinguish three cases for r . If r is not an ancestor or a descendant of r_z , then by Lemma 2.2 no such edge (u, v) exists. Similarly, if r is a descendant of r_z but not on $D[r_z, r_y]$, then by Lemmata 2.2 and 3.6, no such edge (u, v) exists. Finally, suppose that r is an ancestor of r_z , and let u' be the nearest ancestor of u in D_r . Then, u' remains an ancestor of r_z after the insertion, so the shortcut edge in \widehat{G}_r that corresponds to (u, v) does not change. In all three cases the *out* lists of the **IncSCC** structure of \widehat{G}_r remain valid, and hence so does the topological order of its auxiliary components.

It remains to consider the affected auxiliary graphs and \widehat{G}_{r_z} . Let \widehat{G}_r be an affected auxiliary graph with $r \neq r_z$. Then, (i) holds for \widehat{G}_r by Lemmata 3.8 and 3.9. Also, since in \widehat{G}_r we only remove vertices, the topological order for the components left in \widehat{G}_r remains valid. Furthermore, for each scanned canonical vertex w , we delete from $out(f(v))$ any edge (v, w) if $f(v)$ is a canonical vertex in \widehat{G}_r that is not scanned. This implies that (ii) also holds for \widehat{G}_r .

Finally, consider \widehat{G}_{r_z} . From Lemma 3.11, we have that lines 6–9 of **UpdateAC** correctly identify and update the auxiliary components that are merged with components in \widehat{G}_{r_z} . Also, by Lemma 3.10, the remaining auxiliary components that are moved to \widehat{G}_{r_z} are correctly identified in line 5. So, (i) holds for \widehat{G}_{r_z} . To prove that (ii) holds as well, consider a scanned canonical vertex v that is moved to \widehat{G}_{r_z} . By Lemma 3.6, v is a descendant of p , hence \widehat{G}_{r_z} contains a path from $f(p)$ to v . Let w be a vertex that was in \widehat{G}_{r_z} before the insertion of edge (x, y) . Then, Lemma 2.2 implies that if there is a path in $G[D(q)]$ from w to v , then this path contains p . Since v remains a canonical vertex, $v \neq f(p)$ and there is no path from v to $f(p)$. Therefore, the moved auxiliary components are ordered correctly in lines 10–12 of **UpdateAC**, and (ii) follows. \square

3.3 Running time of **SCInc2ECB**

We analyze the running time of Algorithm **SCInc2ECB**. Recall that G is a strongly connected digraph with n vertices that undergoes a sequence of edge insertions. We let m be the total number of edges in G after all insertions ($m \geq n$). First, we bound the time spent by **Initialize**. This procedure is called twice in the beginning of the **SCInc2ECB**, and twice after each time a bridge in G_s or in G_s^R is locally canceled. Then, Lemma 3.5 implies that such an event can happen at most $2(n - 1)$ times. Hence, there are at most $4n$ calls to **Initialize**, and since each execution takes $O(m)$ time, the total time spent on **Initialize** is $O(mn)$. Similarly, the dominator trees of G_s and G_s^R can be updated in total $O(mn)$ time [16]. We next bound the total time required to update the auxiliary components. Consider an execution of **UpdateAC**. Let ν and μ , respectively, be the number of scanned vertices, after the insertion of edge (x, y) , and their adjacent edges. The time to compute the affected subgraph H^* , compute the SCCs of $H^* \setminus r^*$, and the vertices that reach r^* is $O(\nu + \mu)$. In the same time, we can update the auxiliary components of \widehat{G}_{r_z} and of the affected auxiliary graphs, their corresponding topological orders, and the *out* lists of the corresponding **IncSCC** data structures. Since each scanned vertex w is a descendant of an affected vertex, the depth of w decreases by at most one. Hence, the total time spent by **UpdateAC** for all insertions, excluding the execution of line 17, is $O(mn)$. It remains to bound the time required by the **IncSCC** data structures to handle the edge insertions in line 17 of **UpdateAC**. To do this, we extend the analysis from [19]. We say that a vertex v and an edge e are *related* if there is a path that contains both v and e (in any order). Then, there are $O(mn)$ pairs of vertices and edges that can be related in all **IncSCC** structures for every auxiliary graph. We argue that each time the **IncSCC** structure traverses

an edge (after the insertion in line 17 of **UpdateAC**), the cost of this action can be charged to a newly-related vertex-edge pair. Note that we cannot immediately apply the analysis in [19], since here we have the complication that vertices and edges can be inserted to and removed from the **IncSCC** structures. Consider a vertex w and an edge $e = (u, w)$. Call the pair $\langle v, e \rangle$ *active* if v and e are in the same auxiliary graph \hat{G}_r , and *inactive* otherwise. Note that since we identify shortcut edges with their corresponding original edge, e may actually appear in \hat{G}_r as an edge (u', w) , where u' is the nearest ancestor of u in D_r . This fact, however, does not affect our analysis.

Lemma 3.13. *The total number of edge traversals made during the forward searches in all IncSCC data structures is $O(mn)$.*

Proof. To prove the bound, it suffices to show that in all **IncSCC** data structures the total number of unrelated $\langle v, e \rangle$ pairs that are ever created is $O(mn)$. Consider an active pair $\langle v, e \rangle$ that becomes related in \hat{G}_r . Then there is some path π in $G[D(r)]$ that contains both v and e . Suppose that the pair $\langle v, e \rangle$ later becomes active but unrelated in an auxiliary graph $\hat{G}_{r'}$, where r' may be vertex r . Then π does not exist in $G[D(r')]$, which implies that some vertices of π are not descendants of r' . Then, by Lemma 2.2, π must contain the bridge $(d(r'), r')$. Since π exists in $G[D(r)]$, the bridge $(d(r'), r')$ was a descendant of r before some insertion, and then became an ancestor of v . But this is impossible, since after an edge insertion, the new parent $d'(v)$ of v is on the path $D[s, d(v)]$. Hence, once a $\langle v, e \rangle$ pair becomes related, it can never become unrelated. The bound follows. \square

Lemma 3.14. *The total time to update all the IncSCC data structures is $O(mn)$.*

Proof. Updating the lists of out edges in the **IncSCC** data structures, and inserting or deleting canonical vertices can be charged to the cost of updating the dominator tree, and is thus $O(mn)$. By Lemma 3.13, all edge insertions that do not trigger merges of auxiliary components can be handled in $O(mn)$ time. The number of edge insertions that trigger merges of auxiliary components is at most $n - 1$, and each such insertion can be handled in $O(m + n)$ time, excluding unite operations. Taking into account also the total time for all unite operations yields the lemma. \square

Theorem 3.15. *The total running time of Algorithm **SCInc2ECB** for a sequence of edge insertions in a strongly connected digraph with n vertices is $O(mn)$, where m is the total number of edges in G after all insertions.*

4 Incremental algorithm for 2-edge-connected blocks in general graphs

In this section, we show how to extend our algorithm to general digraphs that are not necessarily strongly connected. Let G be input digraph that undergoes edge insertions. We will design a data structure that maintains the 2-edge-connected blocks of G and can report if any two query vertices are 2-edge-connected. We use a two-level data structure. The top level maintains the strongly connected components of G with the use of a **IncSCC** data structure of Section 2.5. We refer to this data structure as **TopIncSCC**.

If the insertion of an edge creates a new component C , algorithm **TopIncSCC** finds the vertices in the new component and updates the condensation of G . Let C_1, C_2, \dots, C_j be the components that were merged into C after the insertion of an edge. We choose the canonical (start) vertex of C to be the start vertex of the largest component C_i . We refer to this component C_i as the *principal component* of C .

Algorithm 4: InsertEdge(G, e)

```
1 Let  $e = (x, y)$ .
2 if  $x$  and  $y$  are in the same component  $C$  then
3   | Execute  $\text{SCInsertEdge}(G[C], e)$ .
4 else
5   | Insert  $e$  into  $\text{TopIncSCC}$ .
6   | if a new component  $C$  is created then
7     | Let  $s$  be the designated start vertex of the largest component merged into  $C$ .
8     | Execute  $\text{Initialize}(G[C], s)$  and  $\text{Initialize}(G^R[C], s)$ .
9   | end
10 end
```

Now we bound the running time of `Inc2ECB`, excluding the time required by `InsertEdge`. The total time required to maintain the `TopIncSCC` structure is $O(mn)$. The total number of new components that can be found by `TopIncSCC` is at most $n - 1$. When such an event occurs, algorithm `Inc2ECB` makes two calls to `Initialize`, and each such call takes $O(m)$ time to initialize the bottom-level structure, i.e., the `SCInc2ECB` data structure for the new component. Hence, the total time required by `Inc2ECB`, excluding the calls to `InsertEdge`, is $O(mn)$.

Next we bound the time spent on calls to `InsertEdge`. First, we bound the total time required to maintain all dominator trees, for each component created by the main algorithm `Inc2ECB`. Recall from Section 3.1 that a vertex v is *scanned* if it is a descendant of an affected vertex. Each scanned vertex v incurs a cost of $O(\text{degree}(v))$, thus we need to bound the number of times a vertex can be scanned. Each time a vertex is scanned, its depth in the dominator tree decreases by at least one. Let C be the current component containing v , and let C' be a new component that C is merged into following an edge insertion. If C is the principal subcomponent of C' then the depth of v may only decrease. Otherwise, the depth of v may increase.

We define the *effective depth* of v after merging C into C' to be zero, if C is the principal subcomponent of C' , and equal to the depth of v in the dominator tree of $G_s[C']$ otherwise. To bound the total amount of work needed to maintain the dominator trees of all components, we compute the sum of the effective depths of v in all the components that v is contained throughout the execution of algorithm `Inc2ECB`. We refer to this sum as the *total effective depth* of v , denoted by $\text{ted}(v)$.

Lemma 4.1. *The total effective depth of any vertex v is $O(n)$.*

Proof. Suppose that the component of v was merged k times as a non-principal component. Let n_i be the number of vertices in the i th component that contains v that was later merged as a non-principal component. The effective depth of v in this component is less than n_i . When a non-principal component is merged, the resulting component has at least $2|C|$ vertices. Thus, the total effective depth of v is $\text{ted}(v) \leq \sum n_i$, where $n_i \leq n$ and $n_{i+1} \geq 2n_i$. To maximize the sum, set $n_k = n$ and $n_i = n_{i+1}/2$, so we have $\text{ted}(v) \leq n + n/2 + \dots + 1 = 2n$. \square

By Lemma 4.1, each vertex v incurs a total cost of $O(n\text{degree}(v))$ while maintaining the dominator trees for all the components of the `TopIncSCC` structure. Hence, the time spent on updating these dominator trees is $O(mn)$. Next, we analyze the total time spent on `Initialize` through calls made by `InsertEdge`. Algorithm `InsertEdge` calls `Initialize` twice for each bridge that is locally canceled, hence at most twice the total number of strong bridges that appear throughout the execution of `Inc2ECB`.

Lemma 4.2. *Throughout the execution of Algorithm Inc2ECB at most $2(n - 1)$ strong bridges can appear.*

Proof. When Algorithm Inc2ECB merges k strongly connected components C_1, C_2, \dots, C_k into a new strongly connected component C , then the new strong bridges that appear in $G[C]$ connect two different components C_i . This is because an edge (u, v) of a subgraph $G[C_i]$ cannot be a strong bridge in $G[C]$ if it was not a strong bridge in $G[C_i]$.

Let H be the multigraph that results from $G[C]$ after contracting each component C_i into a single vertex. We claim that each new strong bridge of (u, v) of $G[C]$ corresponds to a strong bridge (C_i, C_j) in H , where $u \in C_i$ and $v \in C_j$. Multigraph H contains (C_i, C_j) by construction, and there is a unique edge (C_i, C_j) in H . Indeed, if there was another edge (u', v') in $G[C]$ with $u' \in C_i$ and $v' \in C_j$, then (u, v) could not be a strong bridge since $G[C]$ would have a path from u to v , formed by a path from u to u' in $G[C_i]$, edge (u', v') , and a path from v' to v in $G[C_j]$, which avoids (u, v) . If (C_i, C_j) is not a strong bridge in H , then there is a path P in H from C_i to C_j that avoids (C_i, C_j) . Hence, $G[C]$ has at most $2(k - 1)$ new strong bridges.

Thus, we charge at most two new strong bridges each time a component is merged into a larger component. Since there are at most $n - 1$ such merges, the total number of strong bridges that can appear during the sequence of insertions is at most $2(n - 1)$. \square

Hence, by Lemma 4.2, the total time spend on calls `Initialize` via `InsertEdge` is $O(mn)$. Finally, we need to consider the time required to maintain the bottom `BottomIncSCC` structures. Unfortunately, it is no longer true that an active vertex-edge pair that becomes related in a `BottomIncSCC` structure, remains related throughout the execution of the algorithm. However, we can bound the number of times such a pair can change status from active and unrelated to active and related.

Lemma 4.3. *Let v be a vertex, and let e be an edge. The pair $\langle v, e \rangle$ can change status from active and unrelated to active and related at most $\log n$ times.*

Proof. Suppose that $\langle v, e \rangle$ becomes active in the `BottomIncSCC` structure of some strongly connected component $G[C]$ of the top structure. From Lemma 3.13, we have that in order for $\langle v, e \rangle$ to become active but unrelated, C must be merged to another component as a non-principal component. This can happen at most $\log n$ times, so the bound follows. \square

By plugging the above bound in the proof of Lemma 3.14, we get a total bound of $O(mn \log n)$ for maintaining all `BottomIncSCC` structures. We can improve this by employing a more advanced `BottomIncSCC` structure. Namely, we can use the two-way search algorithm of Bender et al. [4]. The algorithm maintains for each canonical vertex v a level $k(v)$. The levels are in a pseudo-topological order, i.e., if (u, v) is an edge (original or formed by some contractions), then $k(f(u)) \leq k(f(v))$. The condensation of G is maintained by storing for each canonical vertex v a list $out(v)$ of the edges (u, w) such that $f(u) = v$, to facilitate forward searches, and also a list $in(v)$ containing vertices w such that $(f(w), v)$ is a loop or an edge of the current condensation with $k(f(w)) = k(v)$, to facilitate backward searches. It sets a parameter $\Delta = \min\{m^{1/2}, n^{2/3}\}$ in order to bound the time spent during a backward search. To insert an edge (x, y) , the algorithm computes $u = f(x)$ and $w = f(y)$. If $u = w$ or $k(u) < k(w)$ then the algorithm terminates. Otherwise, it performs a backward search from u , visiting only canonical vertices at the same level as u . During this search, loops or duplicate edges are not traversed. The backward search ends as soon as it traverses Δ edges or runs out of edges to traverse. If the backward search traverses fewer than Δ edges and $k(w) = k(u)$ then the forward search is not executed. Otherwise, if the backward search traverses Δ edges, or it traverses fewer than Δ edges but $k(w) < k(u)$, then the algorithm executes a forward search from

w . The forward search visits only vertices whose level increases. Finally, if a cycle is detected during the backward or the forward search, the new component is formed.

We use the above algorithm to implement the **BottomIncSCC** data structures. As before, we augment the *out* and *in* lists so that they also store original edges that become shortcut edges. In the following, let $z = nca(x, y)$. All occurrences of an original edge (u, v) , in the list of original edges leaving u , the list of original edges entering v , and possibly in $out(f(u'))$ and $in(f(v))$, where u' is the nearest ancestor of u in D_z , are linked so that we can locate the shortcut edges in constant time. To initialize a **BottomIncSCC** structure for an auxiliary graph \widehat{G}_r (line 13 in procedure **Initialize**), we compute the auxiliary components of \widehat{G}_r , and set the level of each canonical vertex v to be one. Then we create the *out* and *in* lists for the condensation of \widehat{G}_r . In procedure **UpdateAC**, we do not change the levels of the vertices that are not moved. For any vertex that is moved to \widehat{G}_{r_z} , we set its level to be equal to $k(f(p))$. Then we update the *out* and *in* lists (line 16), as in Section 3. Finally, the new edge is added in \widehat{G}_{r_z} (line 17), and we execute the two-way search algorithm of Bender et al. We refer to the implementation of algorithms **SCInc2ECB** and **Inc2ECB**, using the above **BottomIncSCC** data structure, as **SCInc2ECB-B** and **Inc2ECB-B**, respectively.

Lemma 4.4. *Algorithms **SCInc2ECB-B** and **Inc2ECB-B** are correct.*

Proof. The only parts of the algorithms that are affected are the subroutines **Initialize** and **UpdateAC**. In the former, we set the level of each canonical vertex v to be $k(v) = 1$. This is a valid initialization since the levels are in pseudo-topological order. Moreover, lists *out* and *in* contain all the edges in the condensation of an auxiliary graph. Now consider **UpdateAC**. Removing an auxiliary component and its canonical vertex from an auxiliary graph and the **BottomIncSCC** structure, respectively, does not affect the fact that the levels are in pseudo-topological order. Consider now the updates in the **BottomIncSCC** structure of \widehat{G}_{r_z} . In lines 7–9, we merge the auxiliary component of $f(p)$ with some components from other auxiliary graphs. When we do that, we maintain $f(p)$ as the canonical vertex of the formed component, so its level does not change. This means that the levels remain in pseudo-topological order for all canonical vertices that are already in \widehat{G}_{r_z} before the insertion of (x, y) . Consider now the insertion of canonical vertices in the **BottomIncSCC** structure in lines 10–12. The level of all these vertices is set equal to $k(f(p))$. Let v be such a canonical vertex. Let (u, v) be an edge entering v from another canonical vertex of \widehat{G}_{r_z} . Then u is either $f(p)$ or a vertex that was moved to \widehat{G}_{r_z} together with v . In both cases $k(u) = k(v)$. Now let (v, w) be an edge out of v entering another canonical vertex of \widehat{G}_{r_z} . Before the update, \widehat{G}_{r_z} contained a path from $f(p)$ to w , hence $k(f(p)) \leq k(w)$. Thus, the levels remain in pseudo-topological order. \square

To prove the desired $O(mn)$ bound, we extend the analysis of [4]. The analysis requires some additional definitions. An original edge (u, v) is *live* if u and v are in different components and *dead* otherwise. A newly inserted edge that forms a new component is dead. The level of an edge (u, v) is $k(f(u))$ if the edge is live, or equal to its highest level when it was live if the edge is dead. If (u, v) was never live, then it has no level. A component is live if it corresponds to a vertex of the current condensation and dead otherwise. A live component has level equal to the level of its canonical vertex. The level of a dead component is its highest level when it was live. A vertex w and a component C are related if there is a path that contains w and a vertex in C . Also, the number of components, live and dead, is at most $2n - 1$.

Lemma 4.5. *Algorithms **SCInc2ECB-B** and **Inc2ECB-B** run in $O(mn)$ time.*

Proof. It suffices to show that the total time spent by the **BottomIncSCC** data structures is $O(mn)$. The initialization of all such structures takes $O(m)$ time. Since, by Lemma 4.2, the initialization occurs $O(n)$ times, the total time is $O(mn)$. It remains to bound the total insertion time. We show

that the following invariant, used in the analysis of [4, Lemma 4.2], is maintained: For any level $k > 1$ and any level $j < k$, any canonical vertex of level k is related to at least Δ edges of level j and at least $\sqrt{\Delta}$ components of level j . The invariant is true after initialization, since all vertices, edges, and components have level at most one. Bender et al. showed that the invariant is maintained after the insertion of an edge (line 17 of **UpdateAC**), so it remains to show that the invariant is also maintained after the execution of lines 10–12 and 16 of **UpdateAC**.

Let \hat{G}_r be an affected auxiliary graph with $r \neq r_z$, and let v and u be vertices in \hat{G}_r such that v is moved and u is not. Then, v has an affected ancestor t in \hat{G}_r . Let w be a vertex reachable from v in \hat{G}_r . Then w is also reachable from t in \hat{G}_r . We argue that w is moved. Let π be a path from t to w in \hat{G}_r , and let t' be the first vertex on π that is an ancestor of w in D . If $t' = t$ then w is moved. Otherwise, by the parent property of D , $\text{depth}(t') \leq \text{depth}(t)$. The fact that (p, q) (the first bridge on the path from z to y) is not locally canceled and Lemma 3.1 imply that t' is affected. So w is moved in this case as well. Therefore, all vertices in \hat{G}_r that are reachable from v are moved. This means that if v is related to u then it has level at least $k(u)$. Similarly, an edge e that is related to u and is moved has level at most $k(u)$. The invariant holds for u , since the vertices that are moved have level at least equal to the level of u . Hence, the invariant is maintained for \hat{G}_r . Now consider \hat{G}_{r_z} . The vertices in \hat{G}_{r_z} do not change level. This is true also for $f(p)$, since it remains a canonical vertex even if its component is merged with some components from other auxiliary graphs. Similarly, the original edges in \hat{G}_{r_z} also do not change level. Suppose now that e is a shortcut edge $(f(p), w)$ that is deleted and reinserted as a shortcut edge (u, w) , with $u \neq f(p)$. Then u is a moved vertex, so it has level $k(f(p))$. Hence, shortcut edges also do not change level. Notice also that e remains related to all the canonical vertices in \hat{G}_{r_z} it was related before. Indeed, since u is a descendant of p , \hat{G}_{r_z} contains a path from $f(p)$ to u . If before the move there was a path in \hat{G}_{r_z} from a vertex t to e or vice versa, then such a path exists after the move as well. This implies that the invariant holds for the vertices that were already in \hat{G}_{r_z} before the insertion of (x, y) . Finally, consider a moved canonical vertex u . Let v be a vertex related to $f(p)$ with level $k(v) < k(f(p))$. Then, there is a path from v to $f(p)$, so after the move of u , there is a path from v to u . This implies that the invariant holds for u , since it holds for $f(p)$.

Hence, we showed that the invariant is maintained after the execution of lines 10–12 and 16 of **UpdateAC**. By the proof of [4, Lemma 4.2], it is also maintained after the execution of line 17, so overall, subroutine **UpdateAC** maintains the invariant. Hence, as in [4], the maximum level of a vertex is $\min\{m/\Delta, 2n/\sqrt{\Delta}\}$, since for every level other than the maximum, there are at least Δ different edges and $\sqrt{\Delta}$ different components.

So the total time spent by the **BottomIncSCC** data structures, excluding initialization, is $O(\min\{m^{1/2}, n^{2/3}\}m) = O(mn)$. The bounds for **SCInc2ECB-B** and **Inc2ECB-B** follow. \square

Theorem 4.6. *We can maintain the 2-edge-connected blocks of a digraph with n vertices through a sequence of edge insertions in $O(mn)$ time, where m is the total number of edges in G after all insertions.*

5 2-edge-connectivity queries

Here we provide the details of how to use our incremental algorithms for maintaining the 2-edge-connected blocks of Sections 3 and Section 4, in order to answer the following two types of queries:

- (a) Test if two query vertices u and v are 2-edge-connected; if not, report a separating edge for u and v .

(b) Report all the 2-edge-connected blocks.

A separating edge e for u and v is a strong bridge that is contained in all paths from u to v , or in all paths from v to u .

First, we consider queries of type (a). By Lemma 2.6, u and v are 2-edge-connected if and only if they are in the same subtree in the bridge decomposition and they belong to same auxiliary component with respect to both the forward and the reverse flow graphs, G_s and G_s^R . That is, $r_u = r_v$ and $c_u = c_v$ in G_s , and $r_u^R = r_v^R$ and $c_u^R = c_v^R$ in G_s^R . Recall that we keep the auxiliary components in G_s (and similarly in G_s^R) using a disjoint set union data structure [38]. Since we aim at constant time queries, we use such a data structure that can support each *find* operation in worst-case $O(1)$ time and any sequence of *unite* operations in total time $O(n \log n)$ [40]. This way, we can identify the canonical vertex of the auxiliary component containing a query vertex in constant time. Hence, we can test if u and v are 2-edge-connected also in constant time. If u and v are not 2-edge-connected, then we wish to report a corresponding separating edge also in constant time. Suppose first that $r_u \neq r_v$. Without loss of generality, assume that r_u is not a descendant of r_v in D . By Lemma 2.2, the strong bridge $(d(r_v), r_v)$ is a separating edge for u and v . Now consider the case where $r_u = r_v$, but $c_u \neq c_v$. This means u and v are not strongly connected in the induced subgraph $G[D(r_u)]$, and therefore, all paths from c_u to c_v , or all paths from c_v to c_u , use vertices not in $D(r_u)$. Without loss of generality, assume that all paths from c_u to c_v contain a vertex $w \notin D(r_u)$. By Lemma 2.2, all paths from c_w to c_v go through $(d(r_u), r_u)$. Thus, $(d(r_u), r_u)$ is a separating edge for c_u and c_v . We can find a separating edge for u and v when $r_u^R \neq r_v^R$ or $c_u^R \neq c_v^R$ similarly.

We now turn to queries of type (b) and show to report all the 2-edge-connected blocks in optimal $O(n)$ time. For each vertex v we create the label $label(v) = \langle r_x, c_x, r^R, c^R \rangle$, and we insert the pair $\langle label(v), v \rangle$ into a list L . As above, each of the values r_x , c_x , r^R , and c^R is available in $O(1)$ time. Next, we sort the list L lexicographically in $O(n)$ time using bucket sorting. In the sorted list L the vertices of the same 2-edge-connected block appear consecutively, since they have the same label. Thus, all the 2-edge-connected blocks can be reported in $O(n)$ time.

References

- [1] A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014.
- [2] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
- [3] S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.
- [4] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, December 2015.
- [5] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

- [6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.
- [7] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
- [8] C. Demetrescu and G. F. Italiano. Maintaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.
- [9] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.
- [10] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013.
- [11] P. G. Franciosa, G. Gambosi, and U. Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Inf. Process. Lett.*, 61(2):113–120, 1997.
- [12] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.
- [13] H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016.
- [14] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms*, pages 1988–2005, 2015.
- [15] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int’l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015.
- [16] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012.
- [17] L. Georgiadis, G. F. Italiano, and N. Parotsidis. A New Framework for Strong Connectivity and 2-Connectivity in Directed Graphs. *ArXiv e-prints*, November 2015.
- [18] L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015.
- [19] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms*, 8(1):3:1–3:33, January 2012.
- [20] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015)*, 2015.
- [21] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pages 664–672, 1995.
- [22] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–536, 1999.

- [23] M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31(2):364–374, February 2002.
- [24] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.
- [25] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986.
- [26] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012.
- [27] R. Jaber. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015.
- [28] R. Jaber. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016.
- [29] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science, FOCS '99*, pages 81–91, 1999.
- [30] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.
- [31] S. Makino. An algorithm for finding all the k-components of a digraph. *International Journal of Computer Mathematics*, 24(3–4):213–221, 1988.
- [32] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53 – 58, 1996.
- [33] H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76-A.4:513–517, 1993.
- [34] M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proc. 48th IEEE Symposium on Foundations of Computer Science, FOCS '07*, pages 263–271, 2007.
- [35] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.
- [36] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [37] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.
- [38] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [39] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

- [40] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.
- [41] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proc. 32nd ACM Symposium on Theory of Computing*, STOC '00, pages 343–350, 2000.
- [42] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory*,, pages 384–396, 2004.